

AD 740612

RADC-TR-72-11
Technical Report
January 1972

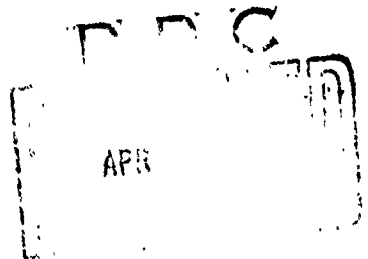


SOME NEW TYPES OF CELLULAR ARRAY COMPUTERS
Polytechnic Institute of Brooklyn

Approved for public release;
distribution unlimited.

NATIONAL TECHNICAL
INFORMATION SERVICE

Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, New York



45

SECTION	<input checked="" type="checkbox"/>
SECTION	<input type="checkbox"/>
SECTION	<input type="checkbox"/>
A	

Do not return this copy. Retain or destroy.

UNCLASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R & D		
(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)		
1. ORIGINATING ACTIVITY (Corporate author) Polytechnic Institute of Brooklyn Route 110 Farmingdale, LI NY 11735		2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED
		2b. GROUP
3. REPORT TITLE SOME NEW TYPES OF CELLULAR ARRAY COMPUTERS		
4. DESCRIPTIVE NOTES (Type of report and inclusive dates) Phase Report		
5. AUTHOR(S) (First name, middle initial, last name) A. E. Laemmel		
6. REPORT DATE January 1972	7a. TOTAL NO OF PAGES 39	7b. NO OF REFS 39
8a. CONTRACT OR GRANT NO F30602-69-C-0053 b. 23 Job Order No. 85050000 c. d.	9a. ORIGINATOR'S REPORT NUMBER(S) PIBEP-71-103 9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report) RADC-TR-72-11	
10. DISTRIBUTION STATEMENT Approved for public release; distribution unlimited.		
11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY Rome Air Development Center (ISCP) Griffiss Air Force Base, New York 13440	
13. ABSTRACT <p>This report describes several new design methods and applications for parallel computers which consist of large arrays of simple cells. A linear memory structure is described which might be used as a compression coder, associative memory, or as a fast memory with slc. components. A very simple cell with a somewhat novel type of computation-universality is applied in a distributed optical processor. Several ways to speed up Turing machines with multiple heads are described.</p>		

DD FORM 1473
1 NOV 65

UNCLASSIFIED

Security Classification

14	KEY WORDS	LINK A		LINK B		LINK C	
		ROLE	WT	ROLE	WT	ROLE	WT
	Computers Parallel Automata Logic Memory Array Turing Machine						

SOME NEW TYPES OF CELLULAR ARRAY COMPUTERS

A. E. Laemmel

Polytechnic Institute of Brooklyn

Approved for public release;
distribution unlimited.

FOREWORD

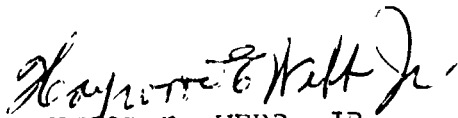
This phase report was prepared under Contract F30602-69-C-0053 by Polytechnic Institute of Brooklyn under Job Order No. 85080000, PIB No. PIBEP-71-103.

Haywood E. Webb, Jr. (ISCP) was the RADC project engineer.

This technical report has been reviewed by the Office of Information (OI) and is releasable to the National Technical Information Service (NTIS).

This technical report has been reviewed and is approved.

Approved:


HAYWOOD E. WEBB, JR
Project Engineer

Approved:


DANIEL R. LORETO, Chief
Computer Technology Branch

ABSTRACT

This report describes several new design methods and applications for parallel computers which consist of large arrays of simple cells. A linear memory structure is described which might be used as a compression coder, associative memory, or as a fast memory with slow components. A very simple cell with a somewhat novel type of computation-universality is applied in a distributed optical processor. Several ways to speed up Turing machines with multiple heads are described.

TABLE OF CONTENTS

1.	Introduction	1
2.	Sequential logic	4
3.	Arrays	10
4.	Optical processor	12
5.	A linear memory array.	16
6.	Multihead Turing machines	22
7.	Variable length coding	31
8.	Appendix 1. Logic Circuit Complexity.	32
9.	References	35

1. INTRODUCTION.

This report describes some of the work on automata and computers, under Contract F 30602-69-C-0053 with the Rome Air Development Center of the Air Force Systems Command. Some previous reports concerning the above contract on these topics are listed as references 1-6, and these contain certain details not repeated here.

The general aim of this report is to consider alternatives to the familiar digital computer architecture. Broadly speaking, practically all digital computers operate sequentially, so that the alternative usually implies some sort of parallel computer. The question of sequential vs. parallel computer has been debated many times, and many different types of parallel computers have been proposed. This report does not purport to cover either the debate or the types of parallel computer, but rather to present some views on the alternatives, some novel design methods for cellular computers, and some examples of applications which appear to have promise of practicality.

Some idea of the spectrum of computer types can be gained from Fig. 1. The arrangement is roughly that greater parallelism, which means more operations going on simultaneously, to the right, and greater efficiency to the top. This arrangement should not be taken too quantitatively. Starting at the upper left, note the digital computer-Turing machine axis. Under digital computers are included all of what might be called the von Neumann model,* ENIAC, 709, 360, PDP8, etc. Of course there are large differences in architecture among these computers, and some parallelism (of input-output, for example), but they are all characterized by their going thru a program step by step. The step carried out by a digital computer is a small step, say adding two integers and fetching the next instruction from memory. If the step is made still smaller, one arrives at the Turing machine,⁽¹⁰⁾ which is shown below digital computers in the spectrum because it is so slow in solving any practical problem. The Turing machine is so slow, in fact, that none has ever been built. A universal Turing machine could be built very cheaply indeed, as they are remarkably simple. All that is needed is a read-write-control head with a few bits of register storage, and a very large supply of tape as the main store.

In Fig. 1 there is a box marked "cellular automata." This refers to a computer which consists of a very large number of very simple cells. The cellular automata are the computers with architecture most different from the usual ones. The diagram suggests that there are two distinct paths from the digital computer-Turing machine region to the cellular automata. The upper path starts at a box marked "computer networks."

* John von Neumann is usually mentioned as the one person who best be singled out for crystallizing the modern digital computer design.^(7, 8) In the present context this is ironic, since he also designed the first cellular computer.⁽⁹⁾

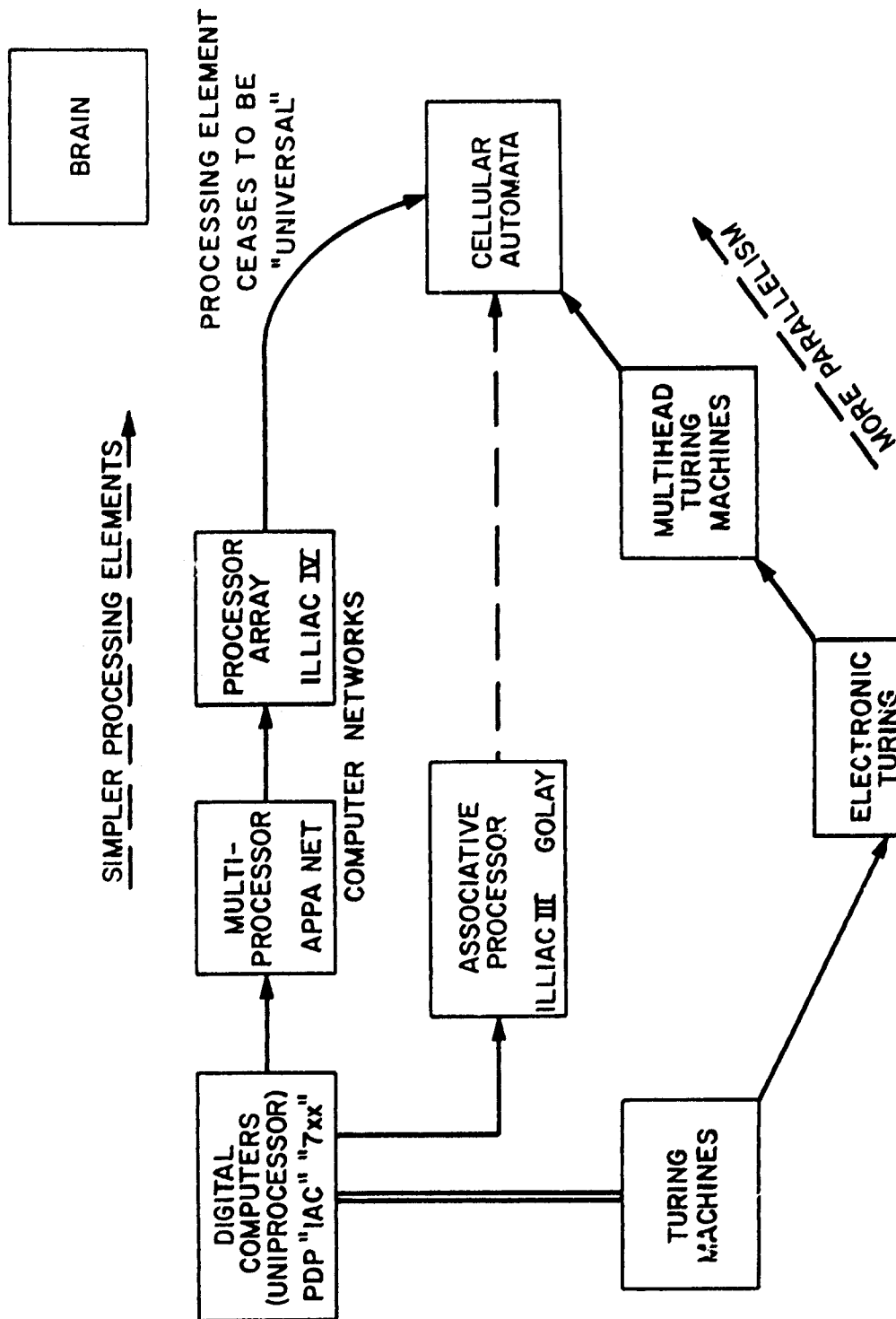


Fig. 1. Spectrum of computer types.

Examples would be the ARPA network,⁽¹¹⁾ or the Illiac IV.⁽¹²⁾ For the same total cost as a single large sequential computer, the individual processing elements would have to be much smaller than what is now considered a large digital computer, say one of the larger 360's; but they still are, in the Illiac IV, about the size of a medium size computer of a few years ago. There is a considerable difference of opinion as to whether it is better to put all of one's resources into a single large computer or an array of smaller computers. For a set of papers championing each approach, see reference 13. One of the principal arguments against parallelism is the difficulty of programming a universal parallel computer, yet one of the pioneers of automatic programming, Grace Hopper, has recently favored the array of smaller computers.⁽¹⁴⁾ The whole question is too involved to even briefly enter here, but it is certainly of some interest to examine what might happen if the trend is continued to some sort of bound, i.e., to make the processing elements simpler and simpler, but more and more numerous. As suggested in Fig. 1, a point will be reached where the processing element ceases to be universal because it is too small. At this point, which will not be sharply defined, some drop in efficiency might occur. Figure 1 pessimistically shows this. On the other hand, for a fixed cost the simpler processing elements could increase the efficiency, so that "cellular automata" might be above "digital computer."

The second path away from the usual sequential types starts out very poorly by going to the all-electronic Turing machine. This is a device in which the tape is replaced by an electronic register and in which the head moves by shifting its state signal along an adjacent register. This structure is then a linear array of cells which each store head state and tape symbol.⁽⁴⁾ Incidentally, this model nicely suggests a physical meaning for Shannon's state-symbol product measure of complexity,⁽¹⁵⁾ the product being the total number of cell states. As suggested in Fig. 1, such an electronic Turing machine would be very inefficient. The cost of the electronic cells would more than make up any increase in speed, so that the electronic Turing machine would be hopelessly impractical. The difficulty is that only one simple cell is active at a time. This is the antithesis of parallel operation. However, the electronic Turing machine does allow multihead operation at no additional cost, and this will increase the efficiency. Several ways to use multiple heads are suggested in Librizzi's thesis⁽⁴⁾ and below in this report. The lower path in Fig. 1 suggests that in the limit where there is a head, i.e., activity, in almost all cells, the cellular automata box is again approached.

Computers on a third path in Fig. 1 deviate from the usual digital computer in that there are several data streams being processed simultaneously, but they differ from other parallel computers in that there is only one instruction stream. If the "instruction stream" only determines the cell state transitions and independently of the problem (as in Section 3), the cellular automata box on the right is again approached.

In the upper right of Fig. 1 is shown the brain. It appears to the right since there must be a large amount of parallelism to operate as it does with very slow neutral or chemical elements. It appears up on the diagram since it is obviously very efficient, having designed all the other computer types. No path leads to it as none has the faintest idea about how it is constructed or how it operates.

In comparing different computer types the criteria of efficiency, cost, etc. must be first established in some quantitative way. Also, it must be decided what class of problems the computer is to work on, how many users are to be served simultaneously, etc. The four boxes in the upper left of Fig. 1 are labelled so as to agree with the terminology of reference 13, wherein will be found comments on which problems each computer seem suited to. As a general rule, parallel computers will show up poorly if the problem and program are simply transferred from a large sequential computer. Parallel computers show up well in at least 4 large classes of problems:

- i) Problems which are inherently parallel in nature and which can be put on a parallel computer in what is almost an analog way. Included here are weather prediction, hydrodynamics, many-particle problems, stress-strain problems, etc.
- ii) Pattern recognition, picture processing, multichannel filtering, and other cases where the data is multidimensional.
- iii) Matrix operations, linear programming, etc.
- iv) Information storage and retrieval, large scale search and trial-and-error methods, and similar problems in artificial intelligence.

2. SEQUENTIAL LOGIC

A paper by Krohn, Maurer, and Rhodes⁽¹⁶⁾ describes an alternative to what they call the "Shannon method" for realizing Boolean functions. The Shannon method is the familiar one using AND, OR, and NOT gates but no memory. Their alternative is to design a fairly simple sequential circuit so that the output at a certain time is the desired Boolean function. The alternative method turns out to be mathematically similar to Maitra cascade,⁽¹⁷⁾ but the hardware is quite different. Actually, the sequential method has been in use for many years; for example, a simple toggle flip-flop can generate the parity function which would require a large number of gates in a zero-memory circuit. Other familiar examples are the counter, serial adder, and accumulator multiplier-divider. As a matter of fact, the whole modern digital computer represents this principle of operating sequentially; it would be possible in theory to make a logic circuit with $10 \times 10 \times 30$ Boolean inputs and outputs which would instantly provide the inverse of a 10×10 matrix, but the complexity would be astronomical. What is being proposed here may sound paradoxical: to make a parallel computer by using

sequential logical elements. The seeming paradox arises because sequential logic circuits gain their simplicity at the expense of operating time, and the whole purpose of the parallel computer is to speed up certain repetitive calculations. It is hoped the resolution of the paradox can be realized if the factor by which logical operations can be paralleled (due to lower element cost) will exceed the factor by which the time of elemental logical operations are increased.

The idea of sequential logic probably first appeared (in cascade form) in the 1962 paper of Maitra.⁽¹⁸⁾ The original Maitra cascade had only one binary rail connecting the elements, and was not capable of realizing an arbitrary Boolean function. Short⁽¹⁹⁾ has shown that a two rail cascade can be designed to realize an arbitrary Boolean function in at least two very simple ways, and his approach will be used here. Yoeli and others have shown that, not only can a two rail cascade realize a pair of arbitrary Boolean functions,⁽²⁰⁾ but an M-ary rail can realize an arbitrary M-ary function of $M \geq 3$.⁽²¹⁾ The design of these more general, and probably more efficient, cascades requires some fairly involved group theory and will not be used in the present report. A trivial generalization of Short's method can realize m arbitrary Boolean functions with $m+1$ rails, i.e., an arbitrary M-ary function with 2 M-ary rail signals. Just provide an extra sum rail for each function required, and add the products into the indicated number of sum rails as they are completed.

The easiest way to design a cascade or sequential logic circuit is to express the desired Boolean function as a sum of products, referring to the Boolean operations of AND and OR respectively. The way this works can be seen at a glance in Fig. 2(a) for the function

$$f(x_1, x_2, x_3) = x_1 x_3' + x_2 x_3'$$

The lower rail accumulates the products, the upper rail accumulates the sum of the products and eventually has the desired function. The first version has two cell types, \otimes and \odot , but as shown in Fig. 2(b) a single type suffices if a control signal is provided to change its function. Since there is now only one type of cell, the possibility of using the same cell over and over suggests itself. This is accomplished by feeding the two rail signals back into the same cell after a small delay as shown in Fig. 2(c). Note that this very simple circuit becomes no more complex no matter how complicated the function f is or how many arguments it has. Of course, the time of operation and the complexity of the control signal do depend on f and the number of its arguments. The arguments which are to be fed into Fig. 2(c) can conveniently be obtained from a shift register as shown in Fig. 2(d). The block A is the circuit shown in Fig. 2(c), except that it is modified to complement the x_i before multiplying it in, if required by the function being realized. In Fig. 2(d) the control is shown operating the two-way shift register to bring the next required x_i under block A. Several variants.

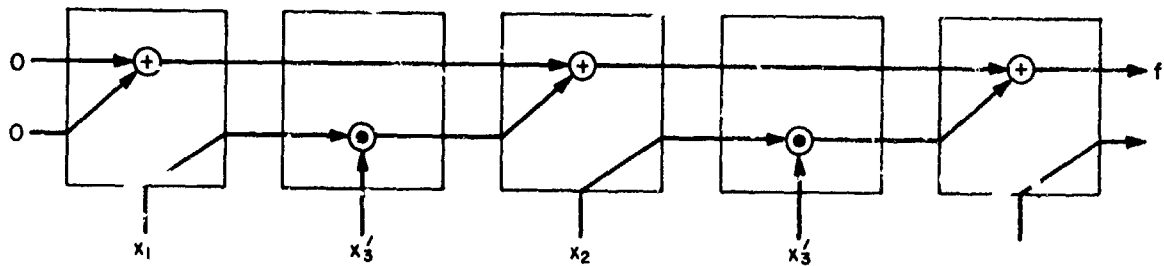


Fig. 2(a). Cascade of 2 types of cell.

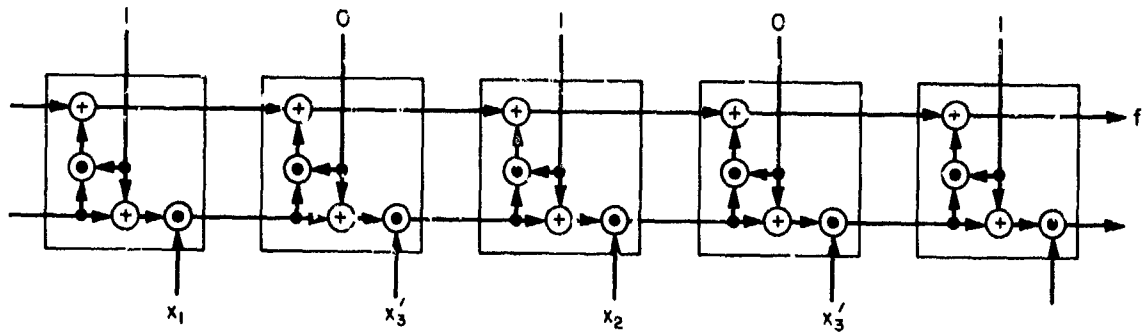


Fig. 2(b). Cascade of identical cells.

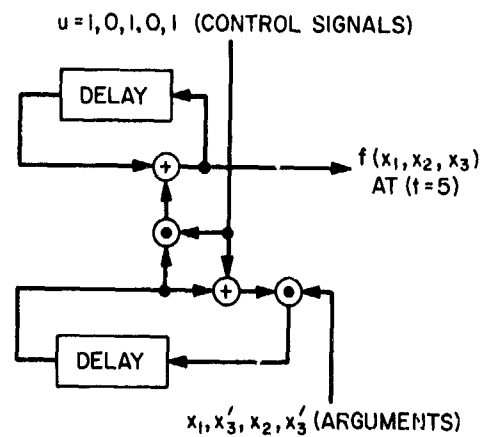


Fig. 2(c). Sequential version of 2(b) above.

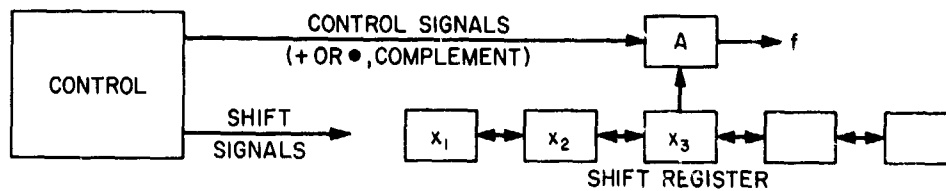


Fig. 2(d). Sequential logic with argument register.

in this process are possible - for example, it could be desired to present all the x_i to A in a regular pattern (shift signals periodic and independent of f). In this case another control signal meaning "ignore the present x_i " could be added, thus permitting the more economical sum-of-products expansion of f rather than the sum-of-minterms which would otherwise be required by the complete periodic shifting.

The shift register in Fig. 2(d) could be a circulating dynamic shift register thus reducing its cost and eliminating the control leads for shifting. If a half adder is put in the circulating register loop the minterms will be generated sequentially thus eliminating the need for the complementation control signal. The control would tell A either to include a minterm or to ignore it, i.e., the control would be the truth table of the desired function f. However, this version would greatly increase the operation time and would not fit in with the applications to be described below.

The time required for the sequential logic to form the desired Boolean function is the critical factor which is needed to compare it to ordinary switching circuits. The complexity of the control will be ignored here because the sequential logic will be used only where it is replicated many times, thus sharing the cost of the control and making it unimportant. The complexity of the sequential element itself is important, since it is desired to minimize the product of operating time and element complexity. Sequential logic operating time is equivalent to cascade length, and the product referred to is equivalent to total cascade complexity. The last statement is true only if the variables can be presented in the required order.

In making estimates of operating time, it is useful to generalize the cascade shown in Fig. 2(a). As shown, the cascade generates a single Boolean function with 2 rails, and has a length which is equal to the number of contacts in a 2 level relay circuit. If $k > 2$ rails are provided, the cascade can realize a single Boolean function with the slightly better efficiency than a k level gate logical circuit (with fan-out limited to 1, i.e., no reentrant paths). If the initialization when starting the function is ignored, and if the variables are available either complemented or not as required, then a set of k cell types will suffice as shown in Fig. 3(a). It might also be observed, as shown in Fig. 3(b), that a k-rail cascade can actually form k-1 independent Boolean functions, but that only one will be a k-level realization, one a (k-1)-level realization, etc. Before considering trees, it might be pointed out that it will sometimes not be necessary to form several functions simultaneously if the cascade cells are allowed to have outputs as well as inputs. This fact, together with the fact that non-binary cell inputs can be handled simply by coding them in binary and using a longer cascade, is illustrated in Fig. 4. A conventional parallel adder is shown at the top as an interactive array of half adders (HA) and full adders (FA). Next, the FA's are realized as 2-HA's and an OR(+). Note that 2 binary rails are required, as indicated by the general theory. The last line shows a circuit to generate carries only. This would be longer if a sum of

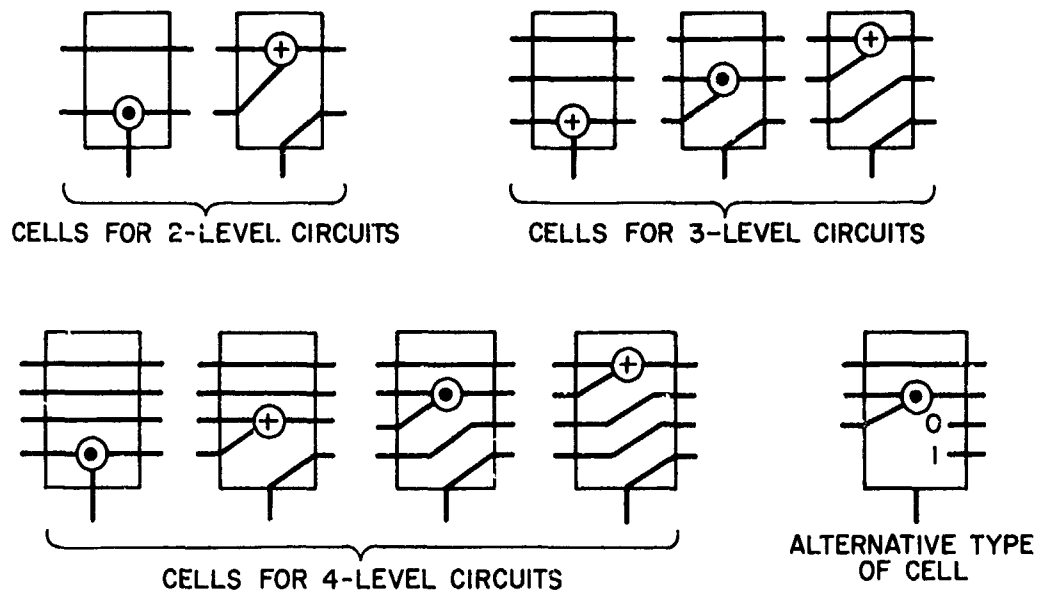


Fig. 3(a). Sets of cell types.

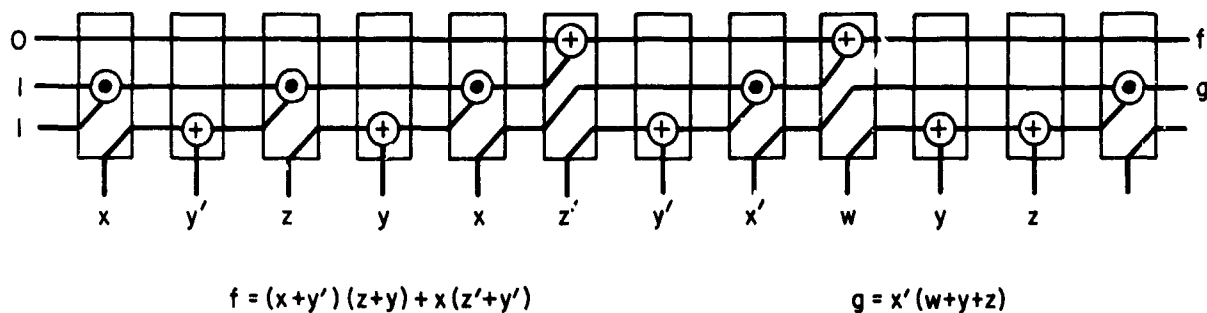


Fig. 3(b). Illustrating 3-level logic (with an extra 2-level function).

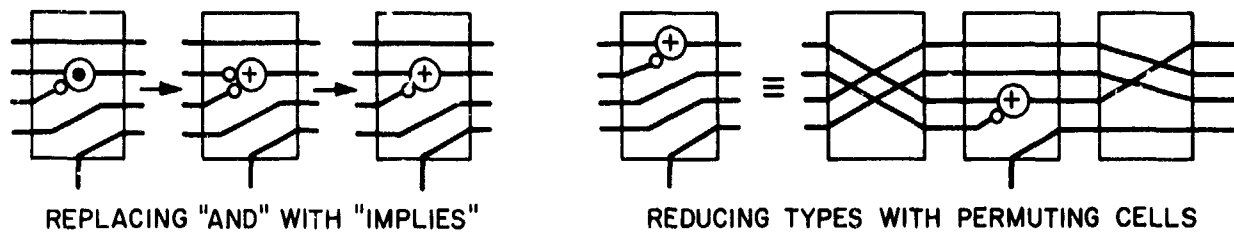


Fig. 3 Cascade versions of logic circuits with more than 2 levels.

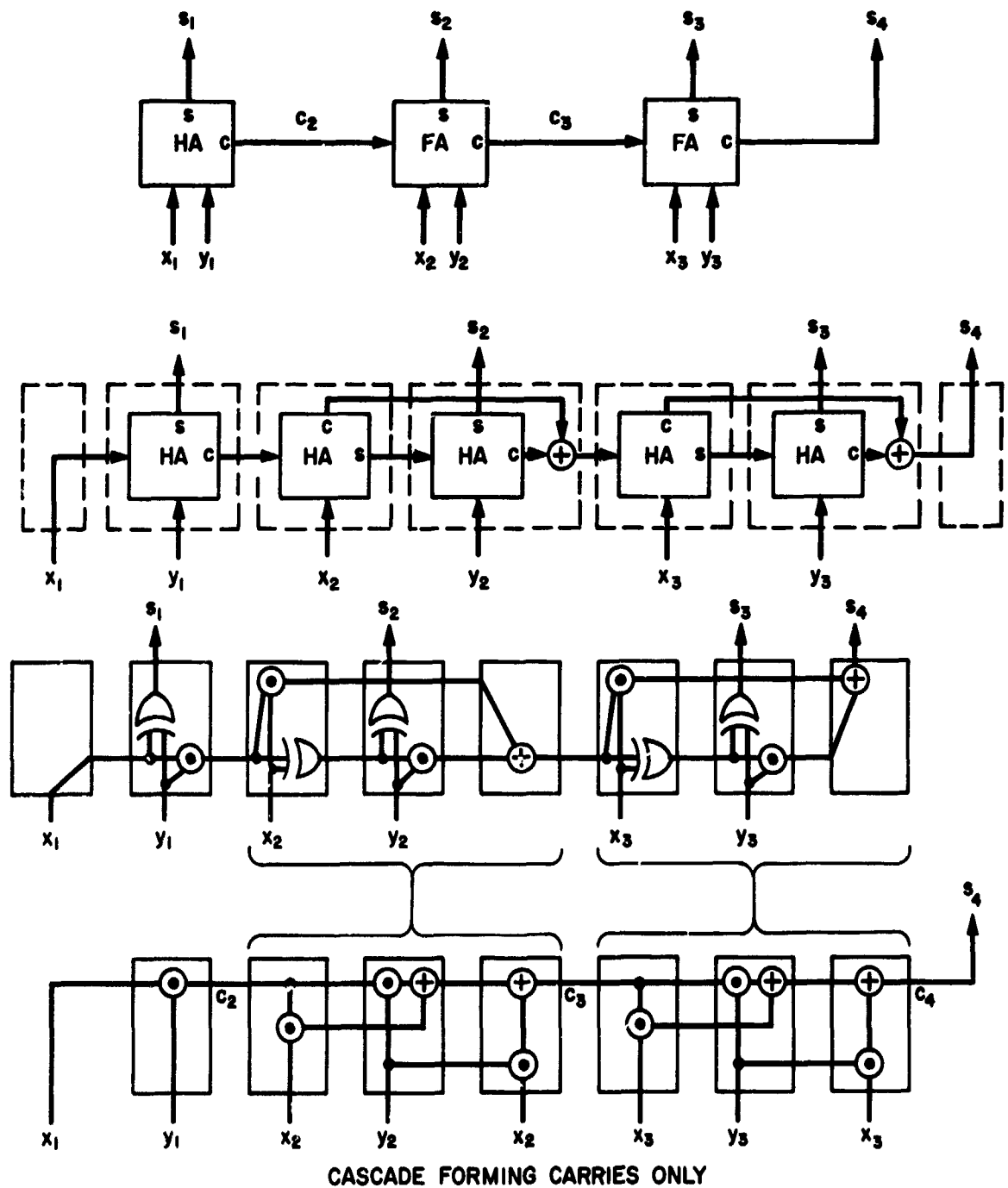


Fig. 4. Comparison of Parallel Adders and Cascades.

products were used, just as a skip carry circuit is more complicated than the iterative ripple carry circuit. Note that the iterative circuit can be repeated indefinitely. Increasing the number of levels (rails) to a fixed number would not accomplish the same end.

The operation time of sequential logic (or the length of a cascade) can either be calculated for the function desired, or as bounds or estimated value for a class of functions. If the class is that of all Boolean functions, the bounds tend to be very pessimistic. The bounds in Appendix 1 are all exponential in the number of variables n , but as was suggested by the adder for a pair of n -digit numbers, the functions of practical interest are more likely to have a linear increase (or at least not exponential, perhaps a power of n). Functions of practical interest usually have symmetries which allow iterative circuits, or else can be decomposed into several simpler functions.

A crude numerical comparison of sequential and combinatorial logic will now be made by reference to Fig. 5(c). This pertains to an array of finite state machines (Section 3) as an application. The cost (time-complexity product) of the combinatorial logic will be $[1][B(n) + F]$, while that of the sequential circuit will be $[2A(n)][16 + 3F]$. These both assume 2-level realizations, $A(n)$ and $B(n)$ are defined in Appendix 1, and F is the cost of a 1 bit store (flip-flop) relative to a gate input. Many assumptions have been made, several of which would cause the sequential circuit to look a little better if they were modified. However, the cost of the sequential circuit is 32 times that of the combinatorial circuit in gate inputs, 6 times in storage elements, and these figures would probably be always greater than one for the type of circuit shown. Why then use sequential logic? There are several possible reasons: The sequential logic is versatile, i. e., the same circuit applies to any function. The initial cost is less for large n , since the second bracket in sequential logic is independent of n . The cost estimates would be much less in some realizations, e. g., the optical processor of Section 4. There might be a gain in reliability.⁽⁶⁾ Also, efficient group-theoretic designs⁽¹⁷⁾ could reduce operating times and generate two functions instead of one.

3. ARRAYS

There is considerable current interest in arrays of logical cells such as the Maitra cascade,^(22, 23) and in arrays of finite state machines,⁽²⁴⁻²⁶⁾ i. e., cells with a memory. A one dimensional array of finite state machines is shown in Fig. 5(a). Each machine consists of a memory register x and a logical next-state circuit F . The next state of the i 'th cell is a given function of its present state x_i and of the present states d neighbors on each side of $x_{i-d}, x_{i-d+1} \dots x_{i-1}, x_{i+1} \dots x_{i+2} \dots x_{i+d}$. The circuit F which calculates this function is identical in each cell (except for its inputs), but it must be repeated in each cell. A sequential logic alternative is shown in Fig. 5(b).

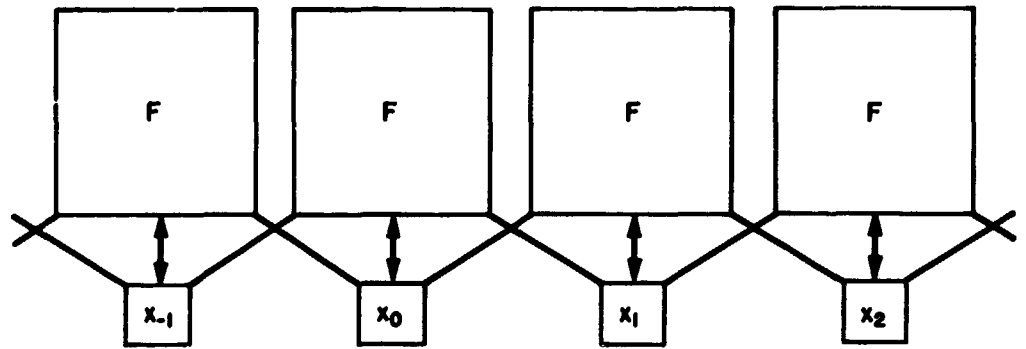


Fig. 5(a). Array of finite state machines.

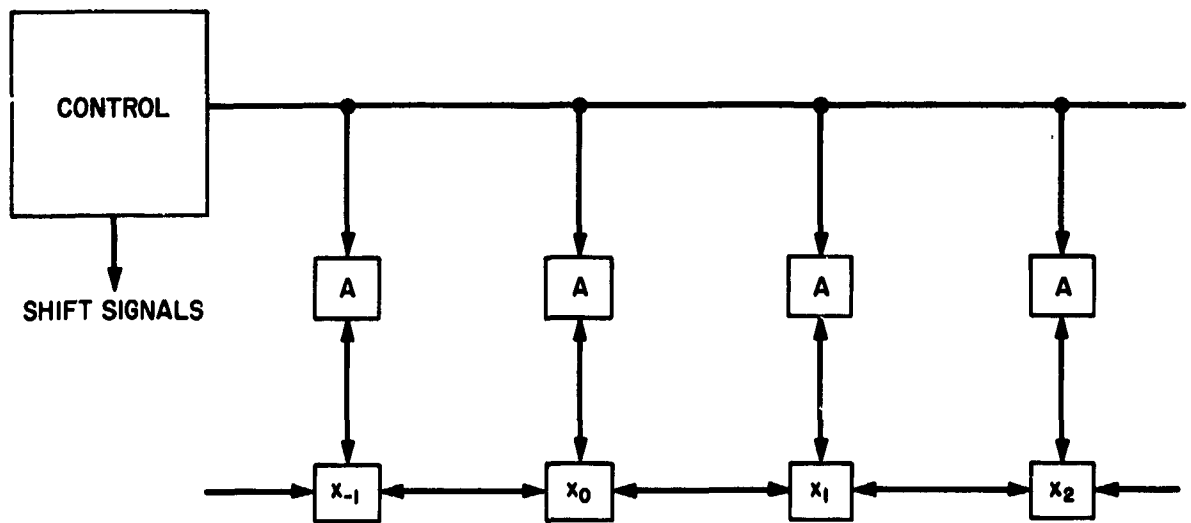


Fig. 5(b). Equivalent array using sequential logic.

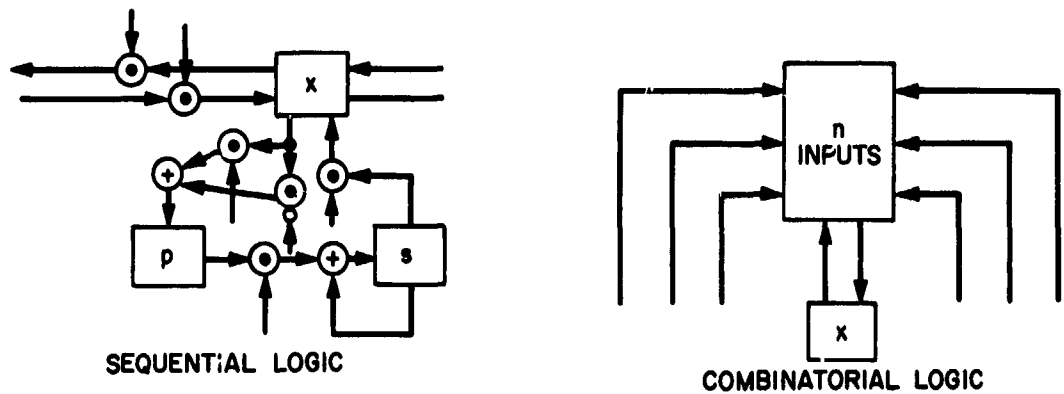


Fig. 5(c). Sequential logic size independent of n .

The only complicated part, the control box, serves all the cells and therefore contributes little to the overall complexity. The cell states are stored in a shift register, and hence there need be no wire communication to distant cells if d is large. Note that the argument shift registers shown in Fig. 2(d) are not duplicated in each cell since a common argument register serves all cells. The x_{i+r} of one cell is the x_{i+r-1} of its neighbor on the right. The shift signals would cause a continual repetition of the pattern $(2d-1)$ right shifts, $(2d-1)$ left shifts, $(2d-1)$ right shifts, etc.

A very desirable feature of Fig. 5(b) is that the array structure is independent of both the next state function $f(x_{i-d} \cdots x_{i+d})$ and of the neighborhood size $2d+1$. It is clearly much easier to modify Fig. 5(b) than Fig. 5(a)! It must again be emphasized that the gains in simplicity and versatility are at the expense of operating time. However, large arrays of the type of Fig. 5(a) are seldom built, but rather a sequential computer is usually used to carry out the function F cell by cell. It is especially against the latter method that the sequential logic array should be able to successfully compete in many applications. More specific circuits for the array will be found in reference 1.

4. OPTICAL PROCESSOR

The need for parallel processing is most urgent with two dimensional signals. The number of resolution elements goes up with the square of the number of resolution lines. Thus, a sequential computer must go over all of the many resolution elements as many times as required in an iterative picture processing process. The time required for this totally sequential calculation makes many applications impractical, especially in real time, even with the very high operating speeds of modern central processing units. Parallel processing has already found useful applications in synthetic aperture radars (where the processing is done optically in analog fashion) and in automatically counting human white blood cells⁽³⁴⁻³⁷⁾ (where the processing is done electronically in digital fashion). Similar applications, where some trials have been made or suggested, are recognition of written or printed text, detection of objects in aerial photographs, improvement of resolution in optical images, solution of two dimensional partial differential equations, "games" imitating life processes, etc.

A two dimensional version of the sequential logic array processor is shown in Fig. 6(a). This looks complicated, but fortunately it is not necessary to make an electronic digital register unit and processing cell for each resolution element. The argument array merely stores the inputs to the logic cells and shifts them so that each cell sees its neighboring arguments in a suitable periodic pattern. The argument array could then be any of the several devices which have been proposed to store optical patterns, and the shifting could be accomplished by projecting the argument array on the cell array with a lens system with an electronically controlled prism to jiggle

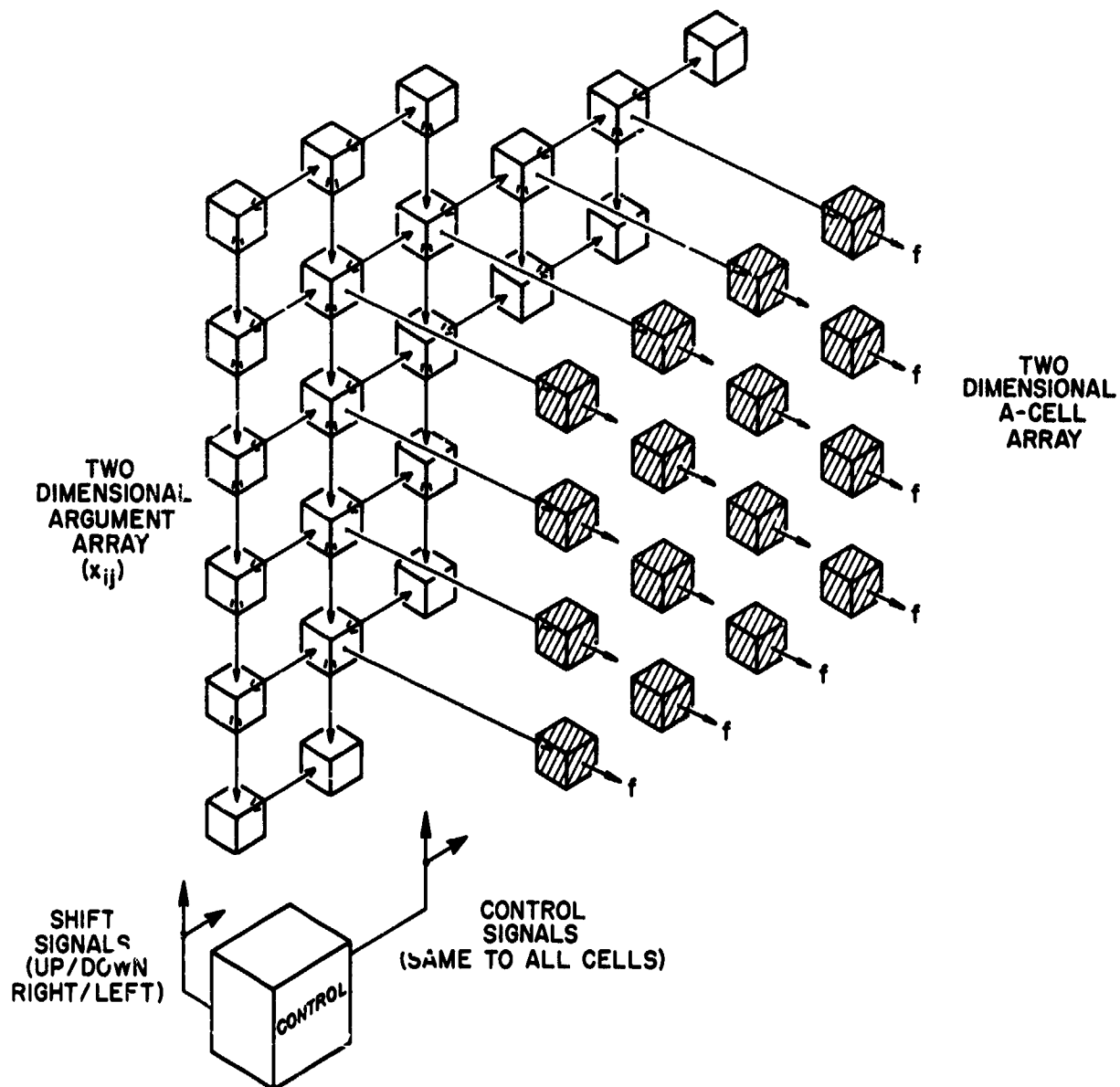


Fig. 6(a). Two dimensional analog of 2(d).

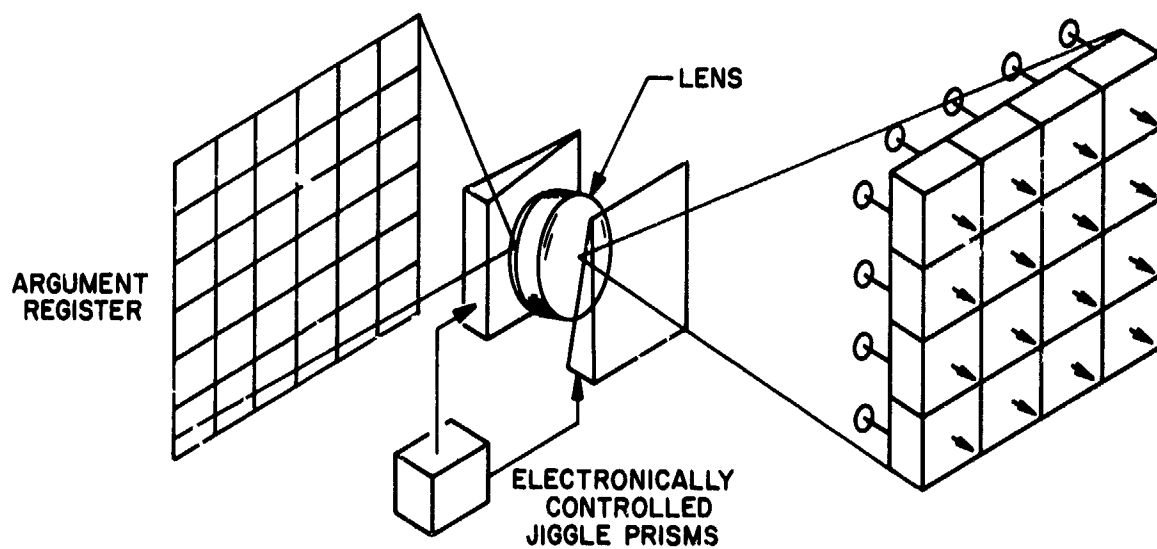


Fig. 6(b). Optical shift register.

the image (two dimensional shifting to neighboring cells). This jiggling is periodic and might be done mechanically by either vibrating the argument register, or mirrors reflecting its image on the processing array, but the speed would not then match the other components. If electron optics are used the shifting would be done by ordinary CRT deflection plates.

In order to see how the processing array of cells can be made, an optical element instead of an array of discrete electronic elements, consider the more detailed version of Fig. 5(b) which is shown in Fig. 7(a). Recall that each A-cell of the former actually has a 2 bit memory corresponding to the two rails of a cascade, the product (p) rail and the sum (s) rail. In the array application of sequential logic the Boolean functions generated (f) are to be returned to the argument register (x), and a set of gates to accomplish this is shown at the top of Fig. 7(a). Note that the \odot gates between the x and p registers, and the \oplus registers between the p and s registers handle multichannel signals under the control of a single control voltage. This suggests using distributed gates over the optical image as is shown in Fig. 7(b). The gate between the p and s registers, on the other hand, simultaneously turns all of the multichannel signals on or off. The latter can be a light valve in the plane of the lens which focuses the image of the p register on the s register. The same applies to the gates which return the s register image to the x register, but in this case a path must be provided around or thru the p register. If a NOT is available, either the AND(\cdot) or the OR(+) can be avoided by de Morgan's laws $xy = (x' + y')'$ and $x + y = (x'y')'$.

The requirements for the distributed optical sequential logic processor will be summarized as follows:

- 1) Means for jiggling a projected image $f(x + \epsilon, y + \delta)$ where ϵ and δ are small periodic functions of time.
- 2) A light valve such that an image $g(x, y) = c f(x, y)$ where $c = 0$ or $c = 1$.
- 3) Means for OR'ing two images in the Boolean sense. If $f(x, y)$ is a stored image it is to be replaced by $f(x, y) + g(x, y)$ where g is the projection of another stored image. This is easy to accomplish by making $f(x, y)$ the latent image on a photographic film, but the slow speed and non-erasability make it unsuitable here.
- 4) Means for AND'ing two images in the Boolean sense, i. e., $f(x, y) \cdot g(x, y)$ over the plane.
- 5) Means for complementing or negating the image in the Boolean sense. High contrast photographic film does this, but again it is too slow and permanent.
- 6) Means for erasing the image. This is necessary when a new value of the function f is to be started. It may similarly be required to make the whole

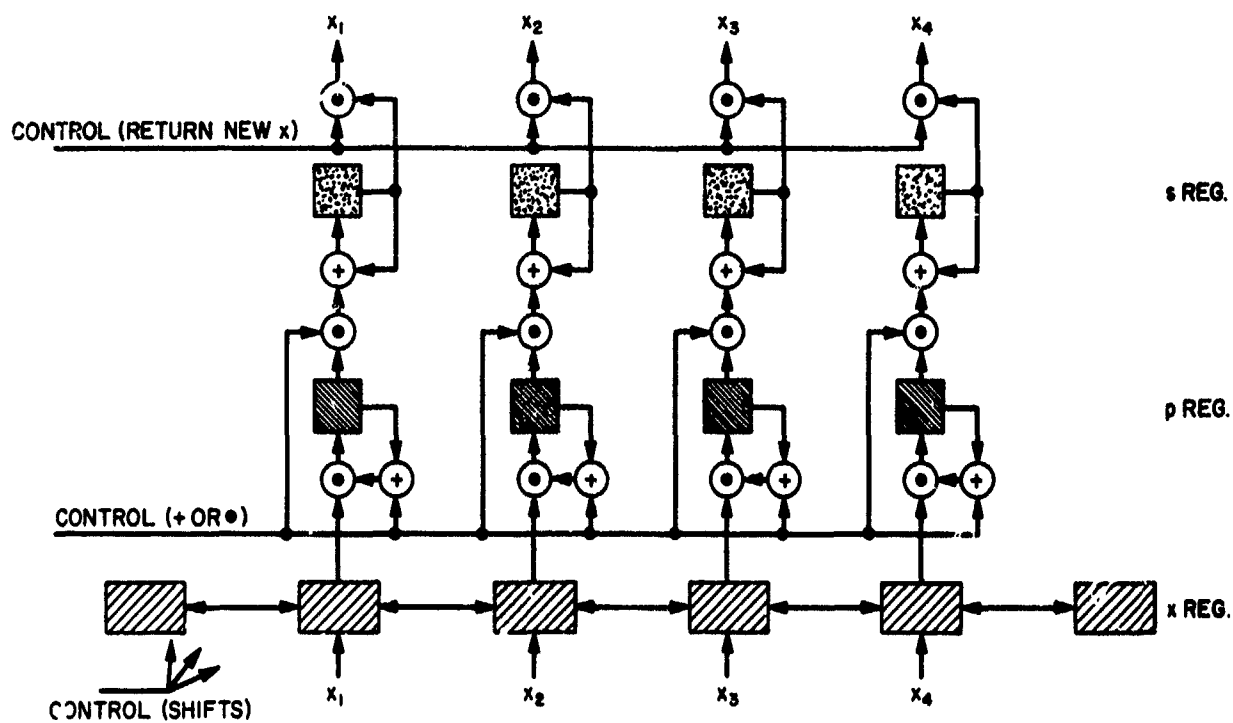


Fig. 7(a). Detailed view of 5(b).

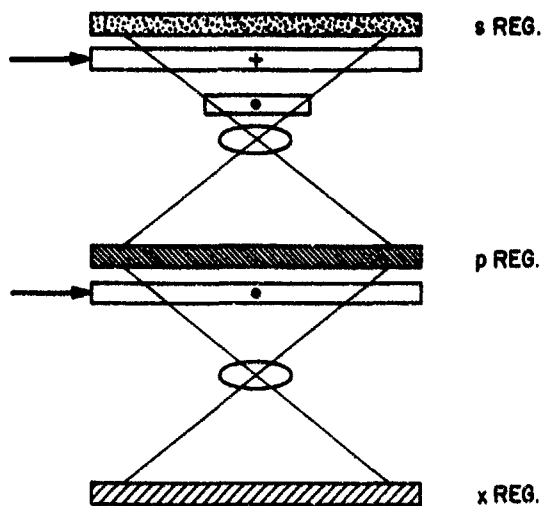


Fig. 7(b). Optical processor using sequential logic.

plane equal to Boolean 1, but this is easily accomplished either by flooding with light or erasing and complementing.

- 7) It also seems desirable to be able to regenerate the images. All of the operations described are needed at a lattice of discrete points $x = x_0n$, $y = y_0m$, and the signals at these points are to be 0 or 1 or some scale. After many operations, errors will accumulate and the point images will spread and overlap. This will eventually cause errors unless the signals are periodically cleaned up.

Not all of the above are required, since it has already been explained that either the OR or the AND can be eliminated by using the other and NOT.

5. A LINEAR MEMORY ARRAY

A novel type of distributed memory has been described by C. Y. Lee⁽²⁸⁾ and several other authors.^(29, 30) The structure to be used here differ from Lee's in that no control sends signals to or receives signals from all cells simultaneously. This permits independent operations in many parts of the array, and might also result in circuits with fewer leads so as to facilitate integrated circuit fabrication.

The basic idea of the linear memory array will be explained briefly. Suppose it is desired to store the table

Name	Item
1	23
12	51
4	1

In the usual computer memory the names would be addresses, would all have the same length, and would not have to be stored explicitly; and the items would be the stored words, and would also all have the same lengths. In the present structure, as illustrated in Fig. 8, the table would be stored in the M register as $\phi 1023\phi 12051\phi 401\phi$. It is desired to find the item whose name is 12, the signal $\phi 120000$ would be fed into the left end of the R register chain and would come out the right and as $\phi 1205100$. The A register is used to keep track of how many digits of the name have been matched, how many digits of the item have been copied out (the "activity" of the Lee memory), and whether the memory is being read or written into ("commands" of the Lee memory).

The details of how the above reading process is carried out can now be followed by the example shown in Fig. 8, and by giving the state transition rules for all registers. The interrogating signal can be seen moving across the bottom of Fig. 8. Note that it is reversed in the physical snapshot because of the left to right propagation. The M

registers do not change during reading. The R registers simply shift from left to right at each transition unless $A_{i-1} = 3$, in which case M_i is shifted into R_i instead of R_{i-1} (but not if $R_i = \phi$). The contents of the A registers are shown as subscripts to the contents of the R registers in Fig. 8, except that the quiescent state $A_i = 0$ is omitted. The A registers are initially 0 and revert to 0 unless otherwise stated. The start of name matching is indicated by $M_i = R_i = \phi$, and this causes A_i to become 1. The condition $A_{i-1} = 1$ causes A_i to become 2 and A_{i-1} to become 0 regardless of the conditions of M_i and R_i . If $A_i = 2$ and $M_i = R_i \neq \phi$, the match continues and A_i becomes 1. The completion of the same matching is indicated by $A_i = 2$ and both M_i and R_i equal to θ , the name-item separating symbol. At this point, A_i becomes 3, which starts the item copying process and also causes A_{i+1} to become 4. The condition $A_{i-1} = 4$ causes A_i to become 3 to continue the copying, unless $M_i = \phi$ indicating the end of the item, or $R_i = \phi$ indicates the end of the space allocated for the item.

So far, the description has covered a read-only memory. There are two ways in which it might be desired to write into a memory with stored names: 1) to replace the item associated with a given name with another item, and 2) to add another name-item combination. The former is illustrated in Fig. 9. The activity register accompanying each symbol to be written in is set to 5. Activity state 5 propagates to the right unless there is a 2 on a 7 to its right, in which case it becomes 6. Activity state 6 propagates to the right just once by changing into a 7 and disappearing. Finally, if $A_i = 7$, on the next transition the contents of R_i are shifted to M_i . The second type of writing involves some additional difficulties and will not be treated until later.

No attempt has been made to minimize the number of states in the memory just described. Each cell would have $N = 7(\alpha+3)^2$ states where the number of symbols in the names and items is α . The 3 comes from the markers ϕ and θ , and the blank 0. The 7 comes from the activity states. If the stored symbols are hexadecimal digits $\alpha = 16$ and $N = 2527 < 2^{12}$, thus it seems that 12 flip-flops are necessary to store 4 bits. The efficiency is not really that bad, because i) the name is stored as well as the item, ii) the names and items can vary in length, thus allowing compression over fixed word length memories, iii) the memory has desirable associative features and more can be easily added, and iv) the memory size is completely flexible in that more capacity can be added without modifying circuits already present.

There is a problem in efficiently packing data in the shifting type of linear memory array. Suppose part of the table is

31	4
123	3115

and the request $\phi 123\theta 000\phi 31\theta 000\phi$ is fed in. With the activity states as described above the output would be $\phi 123\theta 311\phi 31\theta 400\phi$, thus the 5 is lost and two symbols

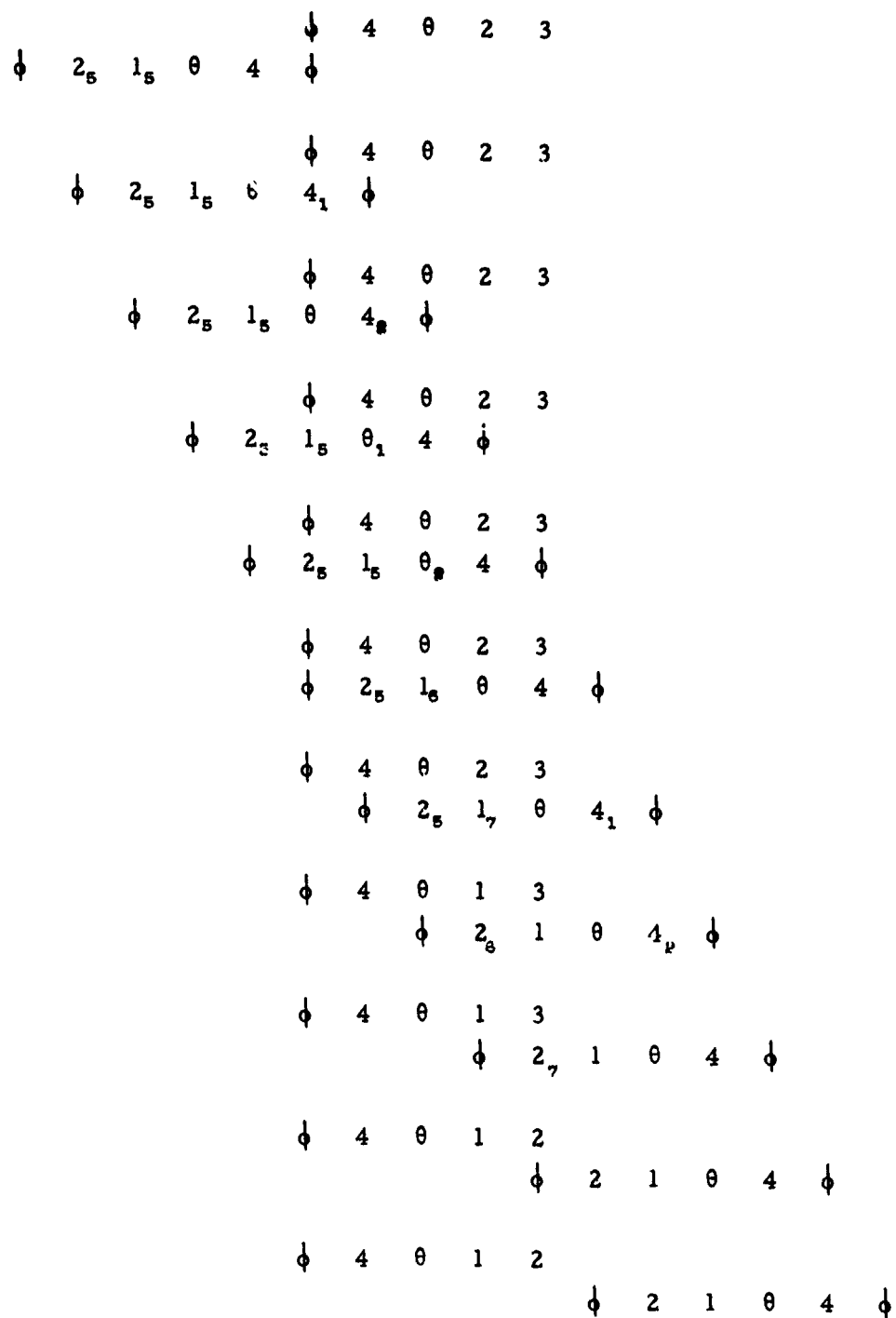


Fig. 9. Writing - Replacing item

after the 4 are wasted. The former is more important since information is lost, yet if the length of the items are not known ahead of time, the only solution seems to be to leave the maximum possible space after each name of the request thus making even more wasted space. It would be nice if extra spaces could be inserted by holding up the string of requests which are following the pertinent one:

ϕ 1 2 3 0 3 1 1 5 ϕ
 ϕ 0 0 0 0 1 3 ϕ 1 1 3
 hold up \longleftarrow

However, this requires that a signal be sent from right to left instantaneously and for an unbounded distance, thus violating the usual definitions of cellular arrays of finite state machines. If such a signal is not sent, in the example above the 3 will try to replace the ϕ , the 1 will try to replace the 3, etc., and one of each pair must be lost.

One way to avoid the need for instantaneous distant signaling in holding up a shift register is to provide blank spaces, i. e., cells in the 0 state, between the symbols being shifted. This is illustrated in Fig. 10(a) where an extra symbol position is created between A and BCD by means not shown. Observe that when BCD are moving, they are spaced two cells apart, when "marking time" they are space one cell apart. The state transition rule is simply that R_i is changed to R_{i-1} only if R_i is initially 0, otherwise R_i remains the same. If the first symbol being held, B here, is allowed to move then the remainder automatically move at the correct time to again provide blank cells. The process resembles a double-ranked shift register in some respects, and the storage efficiency is similarly cut in half. The efficiency could be increased at the expense of communicating to more distant neighbors by bunching the symbols, e. g., AB0CD0..., but the simpler case of alternate blanks will be used here. Holding to avoid errors in reading into insufficient space is shown in Fig. 10(b). Note that in this case an additional advantage is realized: one of the two extra spaces in the block which the 4 was read have been eliminated. This evening-out process would be very useful in implementing a variable length encoding for signal compression.

Three other possible modifications of the linear array memory will be mentioned: First, instead of the name matching ending in failure on one mismatch, with more activity states the match could be called satisfactory with 1, 2 ... errors. Second, the name matching could simultaneously erase the name in the request, but here caution must be exercised so that a partial match will still permit subsequent tries. Third, an economy can be sometimes realized if the name-item combinations are stored in numerical order.

The memory circuits just described would have certain advantages if they were interrogated and read from the same end. It would be easier to make memories whose size is flexible (expandable), frequently used items could be obtained more

		D		C		B		A								
			D		C		B		A							
				D		C		B		A						
					D		C	B			A					
						D	C	B				A				
							D	C		B			A			
								D		C		B			A	
									D		C		B			A

Fig. 10(a). Creation of space without instantaneous signals.

		φ		1		2		3		0		3		1		1		5		φ													
		4		0		1		3		φ				1		3		0		3		2		1		φ							
			4		0		1	3		φ				1		3		0		3		2		1		φ							
				4		0		1	3		φ				1		3		0		3		2		1		φ						
					4		0		1	3		φ				1		3		0		3		2		1		φ					
φ						4		0		1	3		φ			1		1		3		0		3		2		1		φ			
	φ						4		0		1	3		φ			1		1		3		0		3		2		1		φ		
		φ						4		0		1	3		φ			1		1		3		0		3		2		1			
			φ						4		0		1	3		φ			1		1		3		0		3		2		1		
				φ						4		0		1	3		φ			1		1		3		0		3		2			
					φ						4		0		1	3		φ			1		1		3		0		3				
						φ						4		0		1	3		φ			1		1		3		0		3			
							φ						4		0		1	3		φ			1		1		3		0				
								φ						4		0		1	3		φ			1		1		3		0			
									φ						4		0		1	3		φ			1		1		3				
										φ						4		0		1	3		φ			1		1		3			
											φ						4		0		1	3		φ			1		1		3		

Fig. 10(b). Efficient packing of read items of unknown lengths

quickly, and a branching structure to be described below would be easy to implement. In the upper left of Fig. 11, it is shown how to store the name-item combination for straight-thru and reflected reading. The rest of Fig. 11 shows how the name propagates thru a branching memory structure looking for a match. At each junction the name goes both ways, so that the size of the region being searched goes up exponentially with time. If the length between junctions is L bit cells, in t seconds the number of cells searched is

$$N = L \left(1 + 2 + 4 + \dots + 2^{\frac{t}{Lt_0} - 1} \right) = L \left(2^{\frac{t}{Lt_0} - 1} \right)$$

where t_0 is the time required to search 1 cell. Solving for t :

$$t = 3.33 t_0 L \log_{10} \frac{N}{L}$$

As an example, if $t_0 = 1 \mu s$, $L = 100$ and $N = 10^6$, then t is $1.3 \mu s$. Or, consider a "brain" of 10^{10} neurons with $t_0 = 5 \mu s$ and $L = 20$, then t is 2.9 sec. Here is another way in which parallel operation can result in relatively fast overall operation and without broadcast controls. The reflected reading mode would require means to prevent interference between returning items if several simultaneous interrogations are to be allowed. Writing into the branching or dendritic memory could be accomplished by the new name-item combination taking either branch at random each time a junction is reached. The combination would be stored when a sufficiently large empty space was found.

6. MULTIHEAD TURING MACHINES.

The optical processor and linear memory arrays might be regarded as examples of the middle path in Fig. 1. A few ideas on how the lower path might be followed will now be given. More details will be found in Ref. 4. Before describing the Turing machines themselves, some convenient terminology and conventions will be established.

The computer structure of interest here is a linear array of finite state machines such as mentioned above, but without the control box or broadcast leads to all cells. It is important to have a convenient formalism with which to describe the action of such an array, since the mechanism is very unlike conventional computers and many things will go on simultaneously. Roughly speaking, the behavior of the array can be described passively or actively. In the passive description, one says that the state of a cell at time $t+1$ is a function of its state at time t and of the state of its immediate right and left neighbor at time t . This passive description is the most general one, and if each cell has n possible states, it consists of a table of n^3 entries. In the active description one says that a cell in state S_i at time t causes its own state to become S_j and its right neighbor (for example) to become S_k at time $t+1$. This active description cannot be

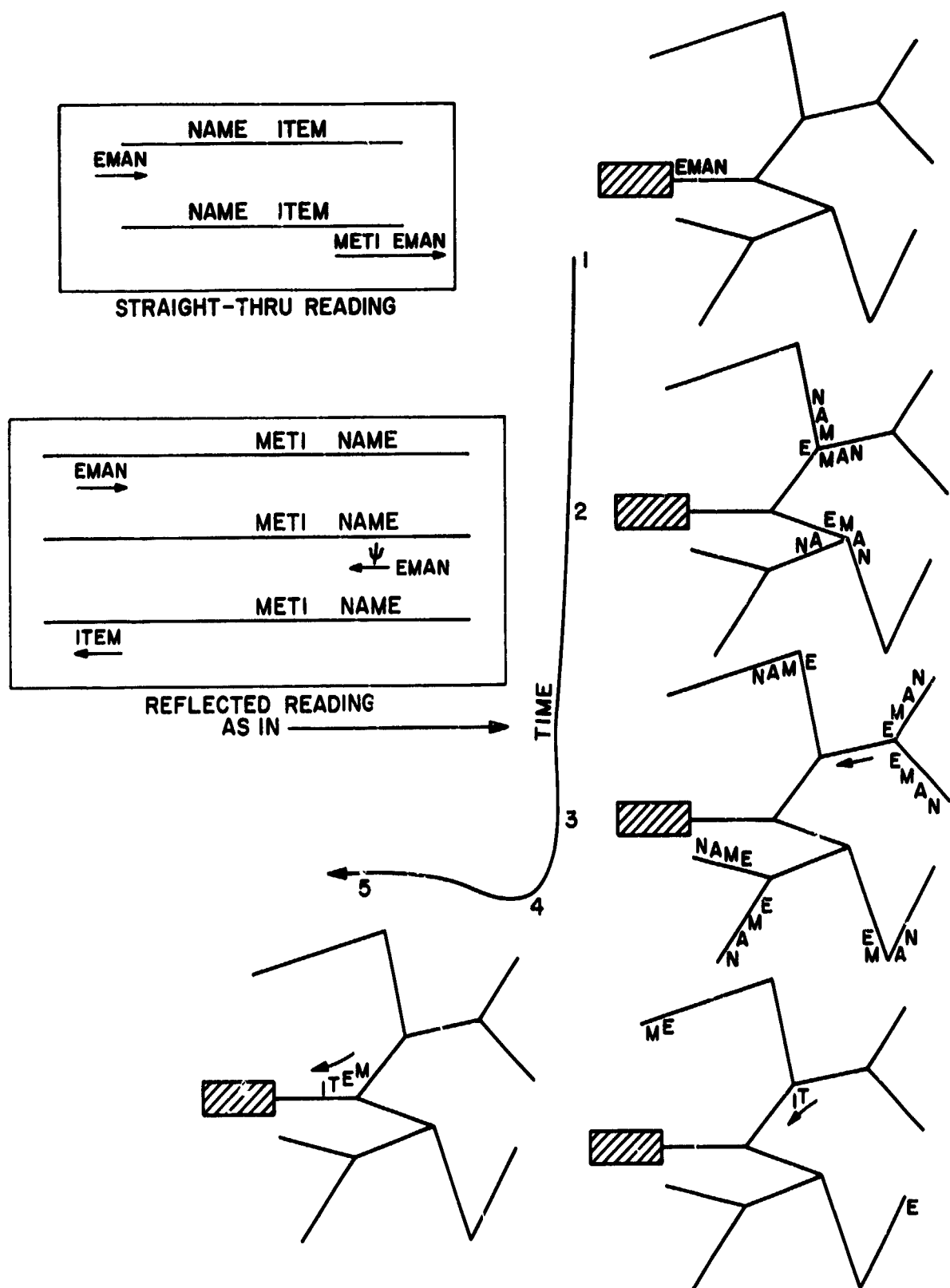


Fig. 11. Dendritic memory.

general since if it consists of a table of only n entries, then the situation where a cell is given contradictory orders by its two neighbors must be avoided. The active description seems natural to describe the action of a Turing machine where a head changes state and moves right or left, or to describe some solutions to the firing squad synchronization problem⁽³⁸⁾ where signals can be visualized moving thru the array. If the active description is augmented by providing a set of rules to resolve conflicts when a cell receives contradictory orders it becomes as general as the passive description, but unless the cell structure is particularized the conflict rules will comprise the bulk of the description.

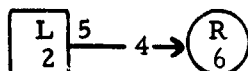
The first particular cell structure will consist of two registers: Q of $(\alpha + 1)$ states and S of σ states, and $n = (\alpha + 1)\sigma$. The register S can be considered as containing one of the symbols $1, 2 \dots \sigma$ which might be written on the tape of an equivalent Turing machine. The register Q can be considered as containing one of the states $1, 2 \dots \alpha$ of the head of the Turing machine, or a 0 representing a tape position with no head there. If the Turing machine has only a single head, only one of the Q registers is non-zero at a time. Now suppose the states of the various cells in the array be represented as a sequence of numbers representing the contents of the S registers with subscripts representing the contents of the Q registers, e. g.

$$\dots 3_0 5_2 1_0 1_0 \dots$$

If the quintuple table of the Turing machine has the entry $(2, 5; 6, 4, R)$, then the next array of states will be

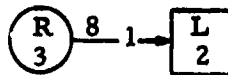
$$\dots 3_0 4_0 1_6 1_0 \dots$$

The above was an active description, a passive description of the same transition would be that the active table of $(\alpha + 1)^3 \sigma^3$ entries contains $(3_0, 5_2, 1_0; 4_0), (5_2, 1_0, 1_0; 1_6)$ among many others. In this case the simpler active description is possible because of three features of the Turing machine which prevent conflicting orders: (i) The new state of the S register depends only on the previous states of the Q and S registers in the same cell. (ii) The new state of the Q register is 0 except possibly when one of its immediate neighbors has a non-0 Q register, and in this case the new Q depends only on the old Q and S of that neighbor, and (iii) There is only one non-0 Q register in the array. Perhaps the clearest way to describe a Turing machine is the state diagram used by Minsky⁽³⁹⁾ in which the above transition would be represented as



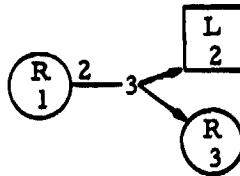
The full convenience of this formalism is demonstrated by a state 3 which moves right until it finds an 8:

$\dots 1_0 3_0 5_3 7_0 1_0 4_0 3_0 8_0 7_0 \dots$
 $\dots 1_0 3_0 5_0 7_3 1_0 4_0 3_0 8_0 7_0 \dots$
 $\dots 1_0 3_0 5_0 7_0 1_3 4_0 3_0 8_0 7_0 \dots$
 $\dots 1_0 3_0 5_0 7_0 1_0 4_3 3_0 8_0 7_0 \dots$
 $\dots 1_0 3_0 5_0 7_0 1_0 4_0 3_3 8_0 7_0 \dots$
 $\dots 1_0 3_0 5_0 7_0 1_0 4_0 3_0 8_3 7_0 \dots$
 $\dots 1_0 3_0 5_0 7_0 1_0 4_0 3_2 1_0 7_0 \dots$



There are Turing machines in which the direction is not a function of the head state alone (independent of the tape symbol) but they can always be replaced by an equivalent "directed state machine."

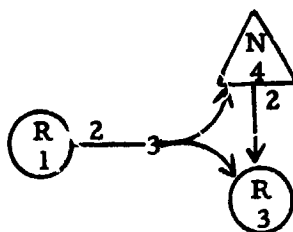
Multiple head Turing machines can be described by slight modification of the above formalism. If two heads are operating simultaneously and independently on remote parts of the same tape no modification of the above diagrams or tables is required. The birth of a new head can be represented by a diagram with two similarly-labeled arrows leading the same state. As an example, the diagram part



might cause the sequence of array states:

$0 \ 1 \ 1_1 \ 0 \ 2 \ 0 \ 1 \ 1 \ 1$
 $0 \ 1 \ 1 \ 0_1 \ 2 \ 0 \ 1 \ 1 \ 1$
 $0 \ 1 \ 1 \ 0 \ 2_1 \ 0 \ 1 \ 1 \ 1$
 $0 \ 1 \ 1 \ 0_2 \ 3 \ 0_3 \ 1 \ 1 \ 1$
 $0 \ 1 \ 1_2 \ 0 \ 3 \ 0 \ 1_3 \ 1 \ 1$

For convenience and clarity the 0-state subscript (indicating no head) has been omitted. It makes it easier to generate a new state moving in the same direction if a "no move" transition indicated by N is permitted. Thus



might then cause

```

0 1 11 0 2 0 1 1 1
0 1 1 01 2 0 1 1 1
0 1 1 0 21 0 1 1 1
0 1 1 0 34 03 1 1 1
0 1 1 0 3 03 13 1 1
0 1 1 0 3 0 13 13 1

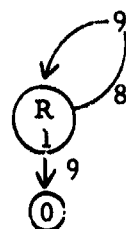
```

Note that the active description needs no clarification here because two heads do not try to move onto the same square simultaneously. It is easy to kill a head simply by making it go to the 0 state. One head out of many can be saved by using a gate which is closed by the first head, e.g.:

```

0 11 0 01 8 0 1 1
0 1 01 0 81 0 1 1
0 1 0 01 9 01 1 1
0 1 0 0 91 0 11 1
0 1 0 0 9 0 1 11

```



If it is desired to have two heads pass each other in opposite directions without interaction there is no difficulty and the diagram need not be augmented if they are separated by an even number of cells:

```

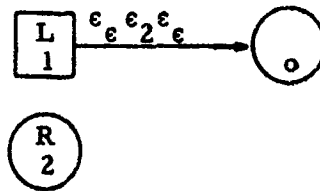
02 1 1 0 1 11
0 12 1 0 11 1
0 1 12 01 1 1
0 1 11 02 1 1
0 11 1 0 12 1

```



The diagram could be augmented if it is desired to have the heads interact. Suppose the 2 state above is to kill the 1 state. This could be shown by putting a triplet label on each arrow such as

0 ₂	1	1	0	1	1 ₁
0	1 ₂	1	0	1 ₁	1
0	1	1 ₂	0 ₁	1	1
0	1	1 ₁	0 ₂	1	1
0	1	1	0	1 ₂	1
0	1	1	0	1	1 ₂



Here ϵ stands for any symbol.

Note that passing heads separated by an odd number of cells are more difficult to deal with under the convention that only neighboring cells can communicate. Thus, in the above example if

0 ₂	1	1	0	1 ₁	1
0	1 ₂	1	0 ₁	1	1
0	1	1 _x	0	1	1

what is x to be? If head 2 is again to kill head 1, then $x = 2$. But if both heads are to succeed independently x can be neither 1 nor 2 nor can x convey the information necessary to preserve both 1 and 2. See Ref. 4 on this point.

Two ways in which multiple heads can speed up the operation of a Turing machine are by carrying out several similar operations simultaneously in different parts of the array, as shown in Fig. 13, and by transferring data in blocks instead of symbol by symbol, as shown in Fig. 14. In either case, the examples show that there is very little increase in the complexity of the state diagram caused by the possibility of additional heads. Left and right moving states are indicated by squares and circles respectively, with a triangle indicating a head which does not move (a unit delay here). Figure 12 shows a unary multiplier from page 125 of Minsky's book.⁽³⁹⁾ It is modified so that when the multiplication is completed the activity moves on to the right looking for another pair of numbers to be multiplied. The multiple-head version shown in Fig. 13 does the same thing except that the activity moves on to the right and simply drops off an extra head to complete any multiplication which has been encountered. Only one extra state is required, and this is the halt state! A new symbol, C , is provided to kill the activity when required, and also (by states not shown) to avoid having left-moving states interfere with the multiplication before it is finished. The latter point is illustrated by the state 17 in the lower right of Fig. 13 being stalled as state 18 until completion of multiplication. A multiple-head word copier is shown in Fig. 14, being essentially Fig. 6.1-6 of Minsky's book.⁽³⁹⁾ The word BBA is converted to states 344 which simultaneously move left and are reconverted to symbols BBA. Only 2 extra

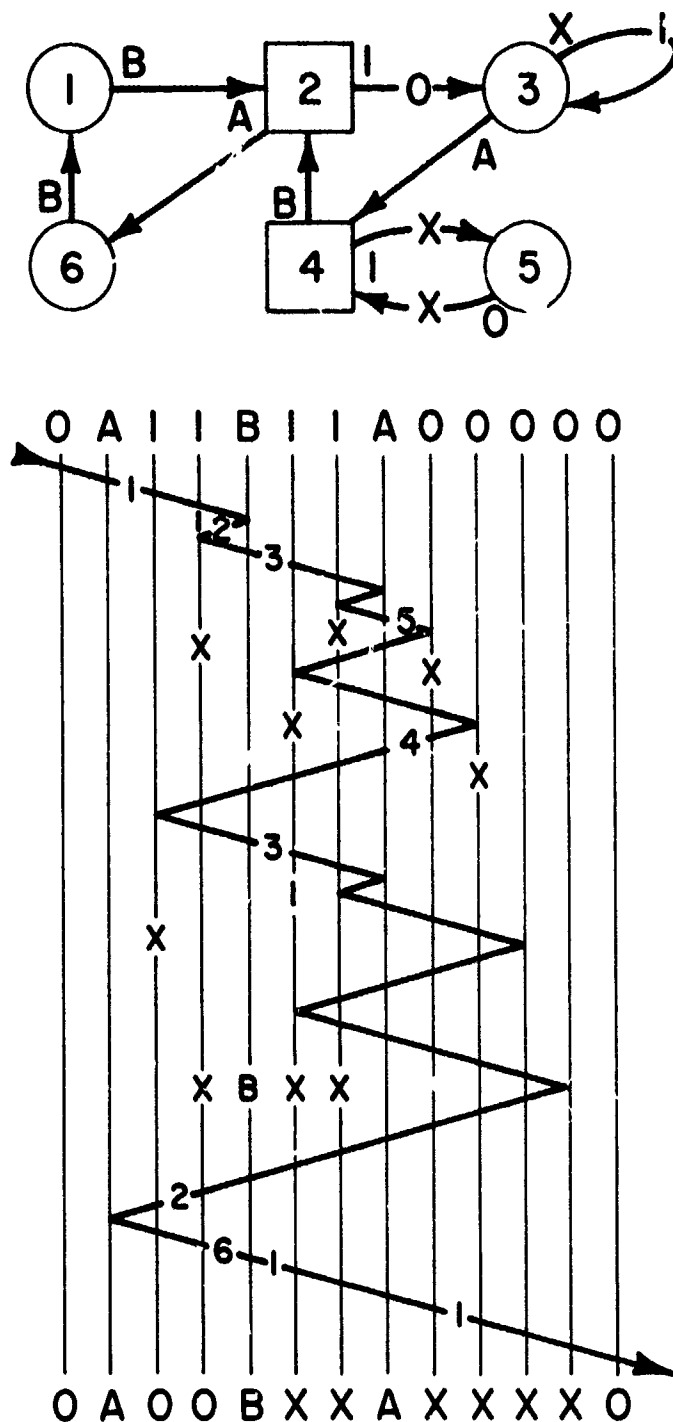


Fig. 12. Unary multiplier Turing machine.

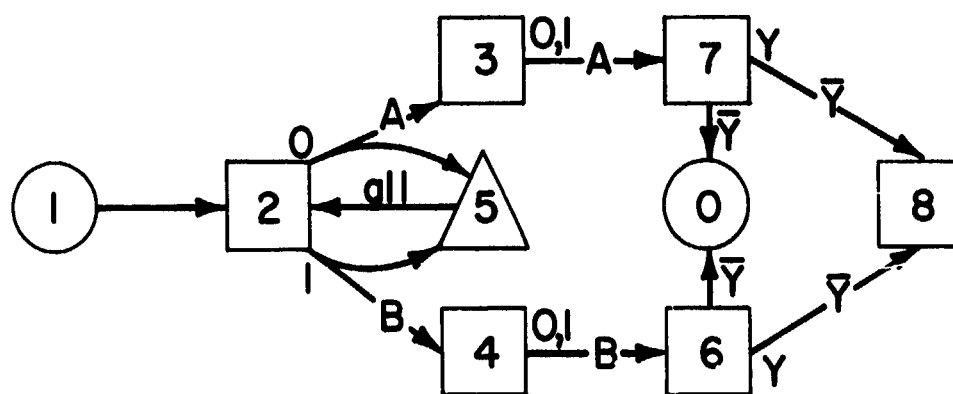


Fig. 14. Multiple-head Turing word copier.

states are required, and one of these is the halt state. Note that the first head to go thru the left writing region, state 7, changes the marker symbol from Y to \bar{Y} , and that \bar{Y} then kills all of the following state 7 heads. This example shows a close connection between multiple-head Turing machines and the memory structures of Section 5 above.

7. VARIABLE LENGTH CODING.

It has been known for a long time that most communications are quite redundant and therefore capable of being considerably compressed. In 1948 the famous theory of Shannon expressed exactly how much compression is possible, and also provided means for designing codes to achieve compression arbitrarily close to the bound. In spite of these facts, little application has been made of compressed transmission of information. It is not so much that the coding apparatus would be complicated, for example the code

$$\begin{array}{l} 00 \rightarrow 0 \\ 01 \rightarrow 10 \\ 1 \rightarrow 11 \end{array}$$

could save almost two times and is extremely simple to implement. The real difficulty is that the calculated compression occurs only on the average over a long time interval. Using the above code, a run of 1's actually is expanded by the code, but this will be compensated by even larger runs of 0's at a later time. The real problem in the application of such codes is in buffering to allow the source and channel rates to be constant while still absorbing temporary fluctuations in the compression ratio. Synchronization problems caused by the variable length words cause additional difficulties if there are channel errors.

The variable length compression codes might provide a very good application for the linear memory array described above in Section 5. The stored name-item combinations would be input-output word combinations of the code table. The rate of reading the memory would be high and the large fixed delay would be of little consequence. The method for creating and filling in gaps illustrated in Fig. 10(b) would even out the fluctuations in compression ratio. The fact that the word pairs can be stored in any order makes the coder quite flexible. The structure with many identical cells, few connections between cells, and few outside connections makes the circuit ideal for integrated circuit fabrication.

APPENDIX 1. LOGIC CIRCUIT COMPLEXITY

The engineering problem of minimizing the complexity of switching circuits is really a modern version of the logical problem of simplifying Boolean expressions. In fact, one of the most frequently used algorithms for the circuit problem was developed by Quine for the logical problem. Some terminology will be established by reference to the function $f(x, y, z)$ which is 1 except when exactly two of its arguments are 1.

$$f(x, y, z) = (x + y + z')(x + y' + z)(x' + y + z)$$

$$f(x, y, z) = xyz + xy'z' + x'yz' + x'y'z + x'y'z'.$$

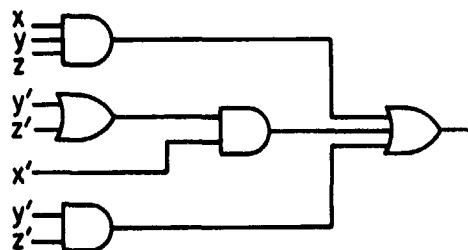
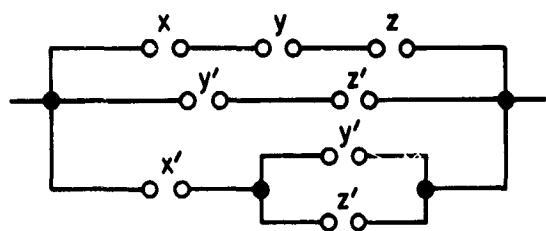
These are the canonical expansions: conjunctive or a product of maxterms, disjunctive or a sum of minterms. If the products of the sum are not required to each contain all variables (i.e., not to be minterms) simpler sum of products can be obtained:

$$f(x, y, z) = xyz + x'y' + y'z' + x'z'.$$

A still simpler expression can be obtained if it is not required that it be a sum of products:

$$f(x, y, z) = x \vee z + x'(y' + z') + y'z'.$$

In order to talk of simplifying the expression for a given function, some measure of complexity is required. Two common measures are A, the number of variable appearances, and B, the number of variable appearances plus the number of terms of more than one factor. If the expressions are realized in the obvious way, then A corresponds to the number of relay contacts, and B corresponds to the number of diodes or gate inputs. For the expressions above, $A = 9, 15, 9$ and 8 , while $B = 12, 20, 13$, and 12 . The circuits for the last expression are



The least upper bound on A and B as functions of the number of variables n and for each type of expression (class of circuit) is important in discussing circuit complexity. The first three expressions will be called 2 level, the last 3 level. In general the number of levels k will be defined as the maximum depth of nested parentheses in the expression if none are omitted, i.e., write $((xy) + (y'z))$ not $xy + y'z$, or as the maximum number of gates a signal must pass thru in the gate circuit. The following table summarizes what is known about the bounds on A and B for various cases:

Expression	Contact Circuit	A	Gate Circuit	B
Disjunctive canonical Sum of min-terms		$n2^n$		$(n+1)2^n$
Sum of products		$n2^{n-1}$		$(n+1)2^{n-1}$
Sum of products of sums		$[2 + (1 - \frac{2}{n}) \cdot (\log_2 n)] 2^{n-1}$		$[3 + (1 - \frac{2}{n}) \cdot (\log_2 n)] 2^{n-1}$
General	Series-parallel 	$3 \cdot 2^{n-1} - 2$		$5 \cdot 2^{n-1} - 4$
General	General 	$\frac{2^{n+1}}{n}$	fan-out = 1	$C_1 \frac{2^n}{\log_2 n}$

Most of the published work on switching circuits in English is concerned with minimization methods and not bounds. The Russians have done much more in the field of general synthesis and bounds, and this is reviewed, together with American work, by Kautz.⁽³¹⁾ The reader is warned against two misprints: the translation of one of Lupanov's articles has $n2^{n-2}$ as the bound for 2-level contact networks instead of $n2^{n-1}$, and Shannon⁽³³⁾ mentions $3 \cdot 2^{n-1} + 2$ instead of $3 \cdot 2^{n-1} - 2$ in correcting his previous statement⁽³²⁾ that this bound is attained by the parity function. Apparently $3 \cdot 2^{n-1} - 2$ is still the best bound known for series-parallel contact networks, but it is not known to be a least upper bound.

No published bounds on 3-level logic elements could be found, the bound shown was derived as follows: Let the function to be expressed in sum of products of sums be $f(x_1, x_2 \dots x_n)$. Expand f as follows:

$$f(x_1 \dots x_n) = x_1 x_2 \dots x_{n-m} f(0, 0 \dots 0, x_{n-m+1} \dots x_n) + x_1' x_2 \dots x_{n-m} \cdot f(1, 0 \dots 0, x_{n-m+1} \dots x_n) + \dots + x_1' x_2' \dots x_{n-m}' f(1, 1 \dots 1, x_{n-m+1} \dots x_n)$$

Next, expand each of the functions on the right as the product of sums which has the least variable appearances. There will be at most $m2^{m-1}$ such appearances in each of the 2^{n-m} functions, and if the $(n-m)2^{n-m}$ appearances of $x_1 \dots x_{n-m}$ as coefficients are added the total is bounded as follows:

$$A \leq 2^{n-m} (n-m + m2^{m-1}) \quad 0 \leq m \leq n$$

As a check, note that if $m = 0$ a minterm expansion is obtained, and if $m = n$ a (dual) sum of products expansion is obtained. For intermediate values of m the value indicated is less than at either end. If the requirement that m be an integer is temporarily dropped, the value of m which minimizes the bound can be obtained by differentiation. The optimum value of m is the solution of the transcendental equation

$$m = \log_2(n-m + \log_2 e) - \log_2(\log_2 e) + 1$$

An approximate solution is $m = \log_2 n$, and inserting this in the above bound gives the value in the table. Note that for large n , increasing the number of levels permitted from 2 to 3 to ∞ decreases the coefficient of 2^{n-1} in the bound on A from n to $\log_2 n$ to 3.

REFERENCES

1. A. E. Laemmel, "General purpose cellular computers," Symposium on Computers and Automata, Polytechnic Institute of Brooklyn, April 13-15, 1971.
2. R. A. Larson, "Algorithms for word problems," Report PIBEP-69-036, Polytechnic Inst. of Brooklyn, June 1969.
3. J. F. Bevaqua, "Logic design for a small computer," Report PIBEP-70-057, Polytechnic Inst. of Brooklyn, 1970.
4. L. Librizzi, "Cellular multihead Turing machine," Report PIBEP-70-058, Polytechnic Inst. of Brooklyn, 1970.
5. A. E. Laemmel, "Parallel computers," pp. 283-6 in Progress Report No. 34 to JSTAC, Report R-452.34-69, Polytechnic Inst. of Brooklyn, Nov. 1969.
6. A. E. Laemmel, "New methods for logical circuit synthesis," pp. 364-8 in Progress Report No. 35 to JSTAC, Report R-452.35-70, Polytechnic Inst. of Brooklyn, Nov. 1970.
7. A. D. Booth and KHV Booth, "Automatic digital calculators," Butterworths, Washington, D. C., 1965.
8. Burks, A. W., H. H. Goldstine and J. von Neumann, "Preliminary discussion of the logical design of an electronic computer," in Collected Works of J. von Neumann and reprinted Datamation, Sept., Oct. 1962.
9. J. von Neumann, "Theory of self-reproducing automata," U. of Illinois Press, Urbana, Ill., 1966.
10. A. M. Turing, "On computable numbers with applications to the Entscheidungs problem," Proc. London Math. Soc., Ser. 2, Vol. 42, pp. 230-65, 1936.
11. P. A. Dickson, "ARPA network will represent integration on a large scale," Electronics Vol. 41, pp. 131-4, 30 Sept. 1968.
12. G. H. Barnes, et al. "The Iliac IV computer," IEEE Trans. Vol. C-17, pp. 747-57, July 1958.
13. G. L. Hollander, "Architecture for large computer systems," Spring Joint Computer Conf., pp. 463-66, 1967 - see also following 4 papers by Andahl, West Fuller and Slotnik.
14. "Hopper against the big machine," Electronics Vol. 44, p. 74, 13 Sept. 1971 - many articles and notes on "Computers in the 70's."
15. C. E. Shannon, "A universal Turing machine with two internal states," pp. 157-165 in "Automata Studies," Princeton U. Press, 1956.
16. K. Krohn, W. D. Maurer and J. Rhodes, "Realizing complex Boolean functions with simple groups," Information and Control, Vol. 9, pp. 190-5, 1966.
17. B. Elspas and H. S. Stone, "Decomposition of group functions and the synthesis of multirail cascades," Conf. Record of the 8th IEEE Sympos. on Switching and Automata Theory, Austin, Tex., Oct. 1967 (pp. 184-196).
18. K. K. Maitra, "Cascaded switching networks of two-input flexible cells," IRE Trans. Vol. EC-11, pp. 136-143, April 1962.

19. R.A. Short, "Two-rail cellular cascades," AFIPS Conf. Proc., Vol. 27, Part 1, pp. 355-369, Spartan Books, Washington, D.C. 1965.
20. M. Yoeli and J. Turner, "Decompositions of group functions with applications to two-rail cascades," Information and Control, Vol. 10, pp. 565-571, 1967.
21. I. Shinahr and M. Yoeli, "Group functions and multivalued cellular cascades," Information and Control, Vol. 15, pp. 369-376, 1969.
22. R.C. Minnick, "A survey of microcellular research," Journ. of Assoc. for Computing Mach., Vol. 14, pp. 203-241, April 1967.
23. W.H. Kautz, "Cellular logic-in-memory array," IEEE Trans. Vol. C-18, pp. 719-727, Aug. 1969.
24. A.W. Burks (ed.), "Essays on cellular automata" U. of Ill. Press, 1970.
25. S.N. Cole, "Real-time computation by n-dimensional iterative arrays of finite-state machines," IEEE Trans. Vol. C-18, pp. 349-65, Apr. 1969.
26. J.N. Sturman, "An iteratively structured general-purpose digital computer," IEEE Trans. Vol. C-17, pp. 2-9, Jan. 1968.
27. R.O. Harger, "Synthetic aperture radar systems" Academic Press, New York, 1970.
28. C.Y. Lee, "Intercommunicating cells, basis for a distributed logic computer," Proc. Fall Joint Computer Conf. pp. 130-136, Dec. 1962.
29. C.Y. Lee and M.C. Paull, "A content addressable distributed logic memory with applications to information retrieval," Proc. IEEE, Vol. 51, pp. 924-32, June 1963.
30. R.S. Gains and C.Y. Lee, "An improved cell memory," IEEE Trans, Vol. EC-14, pp. 72-75. Feb. 1965.
31. W.H. Kautz, "A survey and assessment of progress in switching theory and logical design in the Soviet Union," IEEE Trans. Vol. EC-15, pp. 164-204, April 1966.
32. J. Riordan and C.E. Shannon, "The number of two-terminal series parallel networks," J. Math. Phys., Vol. 21, pp. 83-93, 1942.
33. C.E. Shannon, "The synthesis of two-terminal switching circuits," Bell System Tech. J., Vol. 28, pp. 59-98, Jan. 1949.
34. M.J.E. Golay, "Hexagonal parallel pattern transformations" IEEE Trans. Vol. C-18, pp. 733-40, Aug. 1969.
35. M. Ingran and K. Preston, Jr., "Automatic analysis of blood cells," Sci. Amer. Vol. 223, pp. 72-82, Nov. 1970.
36. K. Preston, Jr., "Use of the Golay logic processor in pattern recognition studies using hexagonal local neighbor logic," Symposium on Computers and Automata, Polytechnic Inst. of Brooklyn, April 13-15, 1971.
37. K. Preston, Jr., "Feature extraction by Golay hexagonal pattern transforms," IEEE Trans. Vol. C-20, pp. 1007-14, Sept. 1971.

38. A. Waksman, "An optimum solution to the firing squad synchronization problem," Info. and Control, Vol. 9, pp. 66-78, 1966.
39. M. Minsky, "Computation: Finite and infinite machines," Prentice Hall, 1967.